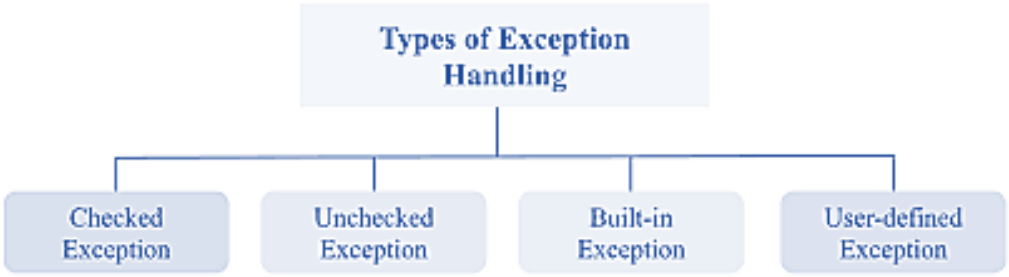
# 11.6-Custom Exception

## Introduction to Custom Exceptions in Java

In Java, the throw keyword is often used to create **custom exceptions**, in addition to handling built-in exceptions (whether checked or unchecked). A **custom exception** (also known as a **user-defined exception**) is a class that extends Java's Exception class. Custom exceptions allow developers to define error conditions that are specific to their application's logic.




Java has a rich set of built-in exceptions that cover most error situations, but there are cases where you might want to throw a more **descriptive**, **application-specific exception.** In such cases, custom exceptions come into play.

## Why Use Custom Exceptions?

Even though Java's built-in exceptions cover many error scenarios, there are times when custom exceptions are useful. Below are a few reasons to use custom exceptions:

- **Business Logic Exceptions**: Custom exceptions are useful for representing errors that are unique to your application's business logic, making them clearer and more descriptive.
- **Better Debugging**: By defining custom exceptions, developers and users can better understand the exact problem that occurred, which makes debugging easier.
- **Categorizing Errors**: Custom exceptions allow you to categorize certain errors and provide specific handling for them.

## How to Create Custom Exceptions in Java

Creating a custom exception involves extending the Exception class (or RuntimeException if it is an unchecked exception). A custom exception can

have its own fields, methods, and constructors. The most common use is to provide a custom error message.

Here's how to create a custom exception in Java:

**Example:**

```
// Custom Exception Class
class NavinException extends Exception {
    public NavinException(String message) {
        super(message); // Passes the error message to the parent Exception class
    }
}

// Main Class
public class Hello {
    public static void main(String[] args) {
        int i = 20, j = 0;

        try {
            j = 18 / i; // Division operation
            if (j == 0) {
                throw new NavinException("Manually thrown exception: Division by zero
not allowed.");
            }
        } catch (NavinException e) {
            j = 18 / 1; // Provide a default value
            System.out.println("Custom Exception Caught: " + e);
        }

        System.out.println("Final value of j: " + j);
        System.out.println("End of Program.. Bye!");
    }
}
```

**Output:**

```
Custom Exception Caught: NavinException: Manually thrown exception: Division by zero
not allowed.
Final value of j: 18
End of Program.. Bye!
```

**Explanation:**

1. **Creating the Custom Exception:**
    - We created a class **NavinException** that extends the **Exception class.**
    - The constructor of NavinException accepts a string message, which is passed to the parent Exception class using the **super() method.** This enables the custom exception to display an error message.

2. **Throwing the Exception:**
    - Inside the try block, we perform a division operation (18 / i). If the value of j is zero (indicating a division by zero), we manually throw a new NavinException with a custom error message.
3. **Catching the Exception:**
    - In the catch block, the custom exception is caught, and we provide a default value for j (18). The message from the exception is printed along with the default value.
4. **End of Program:**
    - The program prints the final value of j and a "goodbye" message.

**Things to Note:**

- The super() method is used to call the parent constructor (Exception) and pass the custom error message to it.
- **Custom exceptions** can be used for specialized error conditions that are not covered by Java's built-in exceptions.
- You can also extend the **RuntimeException** class if you want the custom exception to be **unchecked**.